WORKSHOP INTRODUCTION TO TAPSOLVER

Adam C. Yonge, Dr. Ross Kunz, Dr. Rebecca Fushimi, Dr. AJ Medford

August 2022

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

1.1 What is TAPsolver

Bridging the pressure and material gaps present between top-down and bottom-up approaches to kinetic model construction is an outstanding challenge in the field of catalysis. Deriving mechanistic insights that are satisfactory at all scales and operating conditions remains difficult, but using tools that straddle the line between the two methodologies can help alleviate these issues. Transient kinetic simulations and experiments, especially the TAP reactor, can offer information dense experimental data sets that probe multiple elementary steps and help further constrain previously unobserved elementary processes. Working directly with transient kinetic data can be challenging due to the diversity of initial conditions and mechanisms expressed, as well as the relative complexity of the simulations. TAPsolver is an open-source, Python program for the analysis of TAP reactor experiments. A primary goal of TAPsolver is to make new methods of TAP analysis easily accessible upon publication, leading to community driven development and the unification of workflows. This workshop will show attendees how to simulate TAP experiments and how to analyze the experimental data associated with it, whether it is individual pulses (state-defining experiments) or multi-pulse evolutions (state-altering experiments). It will also help limit the barrier of entry for TAP analysis to facilitate the use of advanced computational techniques by experimental researchers. The python packages FEniCS (Finite Element Computational Software) and Dolfin-Adjoint act as the foundation of TAPsolver and allow for the flexible and efficient application of algorithmic differentiation. TAPsolver was also closely developed alongside TAPSAP, which is used for processing and statistically analyzing raw TAP experimental data.

1.2 Features of TAPsolver

- PDE-based simulations, including state-altering, state-defining and pump-probe operating conditions
- PDE-constrained optimization of kinetic parameters
- Hessian assesment of parameter uncertainties
- Ensemble-based evaluation of initial state/reactor configuration uncertainties
- Model-based design of experiments and model discrimination (under development)
- Efficient kinetic parameter uncertainty propagation to TAP experiments (under development)
- Bayesian parameter estimation (under development)

1.3 Installation

1.3.1 Ubuntu + Mac

Using Conda is the simplest way to install FEniCS (on Ubuntu systems, especially) and is highly recommended. To install the necessary programs for running tapsolver, enter the following lines in the terminal (make sure conda-forge is in your list of channels by running the following commaned: "conda config –append channels conda-forge"):

```
conda create -n tapsolver -c conda-forge/label/cf202003 fenics
conda activate tapsolver
pip install --upgrade git+https://github.com/medford-group/TAPsolver.
                                    git@experimentalFluidity_condensed
pip install --upgrade git+https://github.com/dolfin-adjoint/pyadjoint.
                                    git@faster-ufl
conda install -c anaconda pandas
conda install -c conda-forge/label/cf202003 jsonpickle
```

1.3.2 Windows

FEniCS and Dolfin-Adjoint, the two primary Python packages used to run TAPsolver, are unable to run directly on the Windows operating system. To work with TAPsolver on Windows, we recommend following the instructions outlined by the KMCOS tutorial and setting up Ubuntu on VirtualBox. From there, we recommend setting up Anaconda and installing TAPsolver following the instructions above.

1.3.3 Testing if installation was successful

Once you've installed TAPsolver, you can test that the package is set up correctly by downloading and running the "test install" script available in the github repo.

1.4 Brief overview of TAP experiments

TAP can be a confusing system for those that are new to it. At its core, it is a diffusion based ultra-high vacuum packed bed reactor (PBR). We pulse in a very small number of molecules into one end of this PBR and allow it to diffuse through to the other end of the reactor (outlet). Once the gasses have diffused out of the reactor, another pulse is introduced. This series of pulsing continues on for hundreds or thousands of iterations, allowing investigators to observe how the catalyst material in the center of the reactor "evolves" or changes during the experiment. More details can be found in the literature (particularly https://pubs.rsc.org/en/content/articlehtml/2017/cy/c7cy00678k)

CHAPTER 2
DEFINING A TAPOBJECT

2.1 Overview of the TAPobject

To begin running TAP simulations and working with TAP data, we have to specify a few different aspects. More specifically, we need to specify the reactor and its dimensions, the reactor species in the gas phase and on the surface, and the reaction mechanism and the associated kinetics of the system. All of these components are typically necessary to work with a TAP system.

TAPsolver uses the specification of an object encompassing all of these sub-objects, which is better shown in Figure 2.1. The TAPobject acts as an umbrella, encompassing all of the other objects. Some variables (both necessary and optional) are highlighted in the Figure as well. The general TAPobject variables can also be found in the figure. In the next sections, we are going to step through how to define each of these objects, beginning with the reactor and ending with the TAPobject. This will include a series of code snippets showing how to specifically implement an object in a Python script and will entail the gradual build up to a single TAPobject.

To make the script work properly as you are going through the sections, you must initially import TAPsolver as follows.

```
from tapsolver import *
```

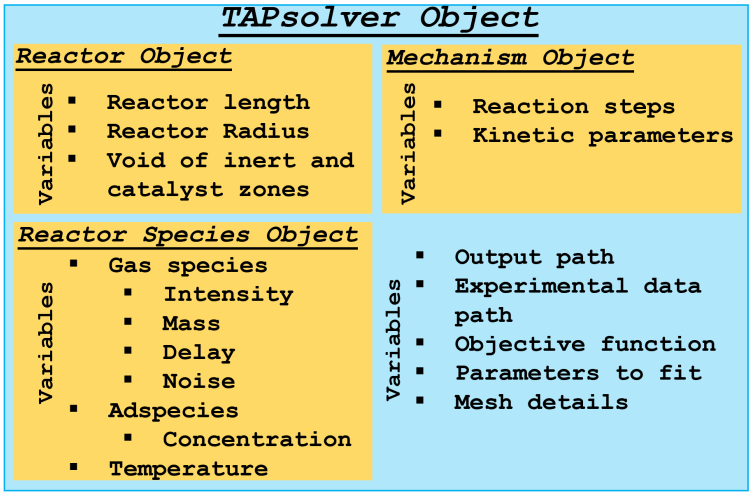

Figure 2.1: The general structure and components of the TAPobject, including the reactor, reactor species, and mechanism objects. These include the variables that are typically defined for the simulation or analysis of TAP experiments within TAPsolver.

2.2 Defining the reactor object

Of all the objects necessary for running a TAP simulation, the reactor is the most straight forward to define. In our example, we will use the default variables for all reactor options. For this reason, we only have to enter the following.

```
new_reactor = reactor()
```

Where the $new\_reactor$ term is our new reactor object, which is initialized by calling reactor(). As mentioned in our introduction to this section, we can control other variables that will be necessary to adjust when working with a real TAP reactor. These variables include

```
## Additional options that may be necessary
# new_reactor.zone_lengths = {0:0.5,1:,0.1,2:0.5}
# new_reactor.zone_voids = {0:0.4,1:0.4,2:0.4}
# new_reactor.reactor_radius = 0.19
```

Some of the variables are defined with a dictionary, including the lengths and voids. In TAP reactors, there are three zones, with two inert zones (indicated by 0 and 2 in the dictionaries) and a catalyst zone (indicated by 1 in the dictionaries). The zone lengths are defined in cm and the voids are unitless. The reactor radius, which is just defined with a single float and is also in units of cm.

2.3 Defining the surface species object

As mentioned in the introduction to this chapter, we define all of the gasses and ad-species (aka surface species, intermediates, adsorbed species) within an object called $reactor\_species$. Similar to the reactor object, we can initialize it by calling the $reactor\_species$ object.

```
species = reactor_species()
```

Our new $reactor\_species$ object is called "species". In the next three subsections, we will show you how to define the reactive gasses, inert gasses, and adspecies.

2.3.1 Adding reactive gasses

Reactive gasses are the primary interest and source of experimental data during TAP experiments. A diverse range of reactants can be introduced into the system at any point and defining them correctly is crucial. Each gas is added one by one to the previously introduced species object. Initially, we define a new gas with the following.

```
CO = define_gas()
```

Where CO (carbon monoxide) is our new gas being defined. With this new gas, we need to specify how much we're pulsing in, the time we're pulsing it in, and the mass of this species. To do this, we add the following

```
CO.mass = 28
CO.intensity = 1
CO.delay = 0.1
```

Where the mass is in amu, the intensity is in nmol, and the delay is in seconds. Last we add the carbon monoxide species to the reactor species object with the add_gas command.

```
species.add_gas('CO',CO)
```

We can add the gasses of oxygen and carbon dioxide following the same steps.

```
O2 = define_gas()
O2.mass = 32
O2.intensity = 1
O2.noise = 0.1
species.add_gas('O2',O2)

CO2 = define_gas()
CO2.mass = 44
CO2.intensity = 0
CO2.noise = 0.1
species.add_gas('CO2',CO2)
```

2.3.2 Adding inert gasses

Inert gasses follow the same rules as the reactive gasses in the previous subsection. For example, argon would be added with the following lines of code.

```
Ar = define_gas()
Ar.mass = 40
Ar.intensity = 1
Ar.noise = 0.1
species.add_inert_gas('Ar',Ar)
```

The only difference is that inert gasses are added with the command add_inert_gas instead of add_gas

2.3.3 Adding adspecies

The workflow for adding adspecies mirrors that of the gasses, with minor deviations around the variables and commands being called. First, we define a new species

```
s = define_adspecies()
```

Again, we are defining a new species "s" through the initialization of $define\_adspecies$. Unlike the gasses, the variables of the adspecies are limited and only entail the specification of the surface concentration (in nmol/cm3).

```
s.concentration = 0
```

Last, we add this new adspecies to the "species" object.

```
species.add_adspecies('CO*',s)
```

Where 'CO*' is adsorbed carbon monoxide. We can add other adspecies in the same way.

```
s = define_adspecies()
s.concentration = 0
species.add_adspecies('O*',s)

s = define_adspecies()
s.concentration = 100
species.add_adspecies('*',s)
```

Here, '*' is the active or open site and adsorbed atomic oxygen is defined as 'CO*'. With these specifications, we have been able to define all gas and adspecies for a TAP simulation.

2.4 Defining the mechanism object

Now that the adspecies and gasses have been defined, we can begin defining the elementary processes and the rate constants. First we make our mechanism object

```
mech = mechanism()
```

Our initialized mechanism object is called "mech", and we will add each of the elementary steps and rate constants to this mechanism object. To define a new elementary step, we use the command

```
mech.elementary_processes[0] = elementary_process('CO + * <-> CO*')
```

Where we are adding the '0'th reaction step to our mechanism, which is '$CO+*<->CO*$'. All reactions steps follow this format, i.e. '$[species1]+[species2]<->[species3]+[species4]$'. TAPsolver takes these steps and converts them into the necessary stoichiometric matrix for the PDEs to use. Upon writing the elementary step, we are able to add kinetic parameters in the forward and backward directions with the following commands.

```
mech.elementary_processes[0].forward.k = 1
mech.elementary_processes[0].backward.k = 1
```

And we can add additional elementary steps in a similar fashion with the following.

```
mech.elementary_processes[1] = elementary_process('O2 + 2* <-> 2O*')
mech.elementary_processes[1].forward.k = 1
mech.elementary_processes[1].backward.k = 1

mech.elementary_processes[2] = elementary_process('CO* + O* <-> CO2 + 2*
                                        ')
mech.elementary_processes[2].forward.k = 1
mech.elementary_processes[2].backward.k = 0
```

It is important to note the slight variation in the specification of the processes. For example, we are including '2' in front of '*' indicates that we are consuming two active sites in the dissociative adsorption of O2. This two, or any other integer, is handled by TAPsolver. Last, to "compile" the mechanism object, we must run the following commands.

```
for j in mech.elementary_processes:
  mech.elementary_processes[j].forward.use = 'k'
  try:
    mech.elementary_processes[j].backward.use = 'k'
  except:
    pass

mechanism_constructor(mech)
```

2.5 The TAPobject

Finally, we will define the TAPobject and add all of the components we just generated. First, we generate a new TAPobject named "simulation". We then add each of the objects to the "simulation".

```
simulation = TAPobject()
simulation.reactor = new_reactor
simulation.reactor_species = species
simulation.mechanism = mech
```

The final import thing to remember about the TAPobject is that it can be saved and read again for later use. This is achievable with

```
save_object(simulation,'./mech_1.json')
mech_1 = read_TAPobject('./mech_1.json')
```

2.6 Practice problems

2.6.1 Problem 1

Write a $reactor\_species$ object with the following species Gas 1: NH_3

- Intensity = 2.5 nmol
- Delay = 0.1 seconds

Adspecies 1: NH_3*

- concentration = 0 nmol/cm3

Adspecies 2: $*$

- concentration = 100 nmol/cm3

2.6.2 Problem 2

Convert the mechanism below into the appropriate code that TAPsolver can use

- $NH_3 + * \leftrightarrow NH_3*$
- $NH_3 * + * \leftrightarrow NH_2 * + H*$

2.6.3 Problem 3

Specify the kinetics of the steps in Problem 2 in TAPsolver form, if

- k_1^f = 1
- k_1^b = 2
- k_2^f = 3
- k_2^b = 4

CHAPTER 3
THE FORWARD PROBLEM

Now that we've learned how to set up a TAPsolver object we can start running simulations and visualizing the results. We will use the TAPsolver object generated in the previous chapter to run these simulations. We will be pulsing in carbon monoxide and oxygen and observing their consumption and the generation of carbon dioxide. We will also run multipulse simulations and offer some additional practice problems.

3.1 State defining experiments

Relative to the construction of a TAPobject, running simulations is straight forward and primarily depends on calling the $forward\_problem$ function.

```
forward_problem(1,1,simulation)
```

In this function call, we are providing three inputs. First, we define the pulse time, or how long we wait before pulsing in additional reactive or inert gasses. In the example, we say that we want to run a one second pulse. Second, we define the number of pulses that we'd like to simulate. Again, we choose to only have a single pulse in our simulation. Last, we specify the TAPobject (here named "simulation") to use in our simulation. When we run this function, we will be returned the following.

```
forward_problem(1,1,simulation)

Warning: Catalyst zone will be refined and rounded to the nearest whole mesh point!

New Catalyst Fraction = 0.01
Old Catalyst Fraction = 0.009900990099009875
Change = 1.0%

Pulse #: 0
Percent: [------------------->] 99.9%Completed in: 16.455 seconds
```

Figure 3.1

The text displayed on the prompt during the simulation show any minor adjustments made to the mesh to accommodate the catalyst zone length and also the progress of simulation. Having a data file for future use is nice, but actually visualizing it can be even more useful. To generate a graph with the flux curves, the following function can be called.

```
flux_graph(simulation)
```

And running this code generates the following graph

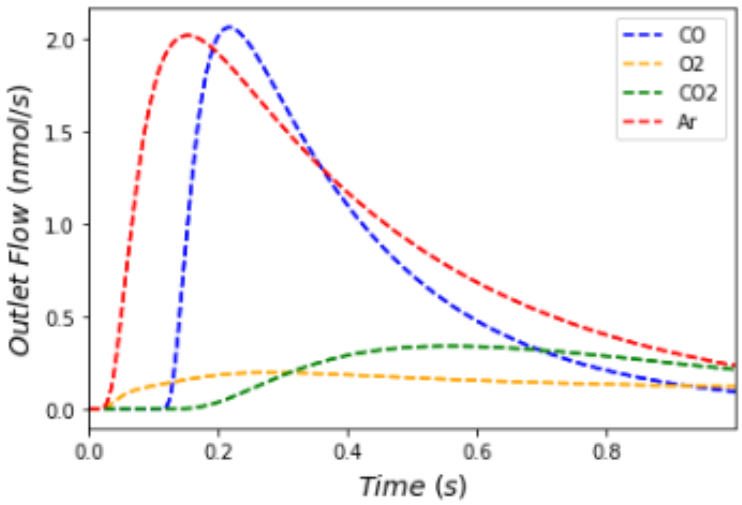

Figure 3.2

We can also vary the intensity and rerun the example as follows.

```
simulation.reactor_species.gasses['CO'].intensity = 2
forward_problem(1,1,simulation)
```

When running the simulation with these adjusted initial conditions, the graph is shifted into.

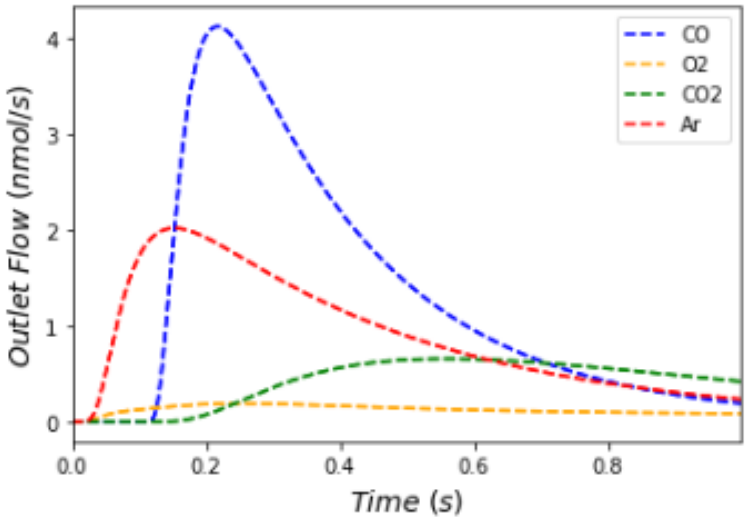

Figure 3.3

Similarly, we can vary the delay of the reactive species and plot the result.

```
simulation.reactor_species.gasses['CO'].delay = 0.1
forward_problem(1,1,simulation)
```

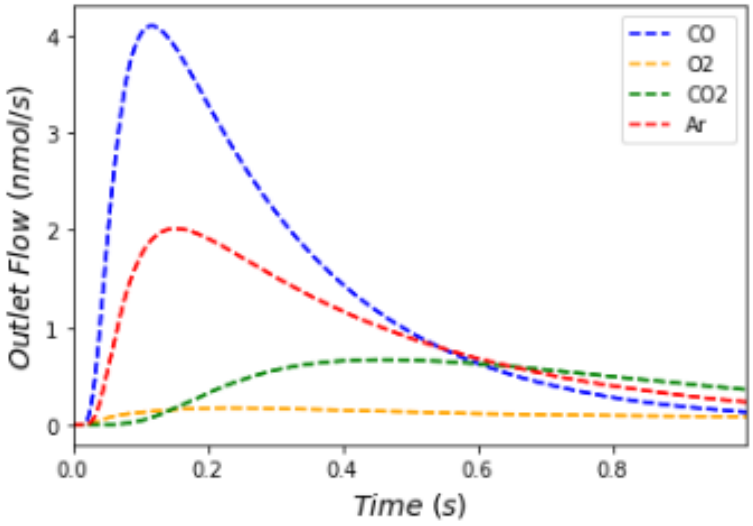

Figure 3.4

Last, we can also incorporate synthetic noise into the system by running the following command.

```
simulation.reactor_species.gasses['CO'].noise = 0.1
simulation.reactor_species.gasses['O2'].noise = 0.1
simulation.reactor_species.gasses['CO2'].noise = 0.1
simulation.reactor_species.inert_gasses['Ar'].noise = 0.1
forward_problem(1,1,simulation)
```

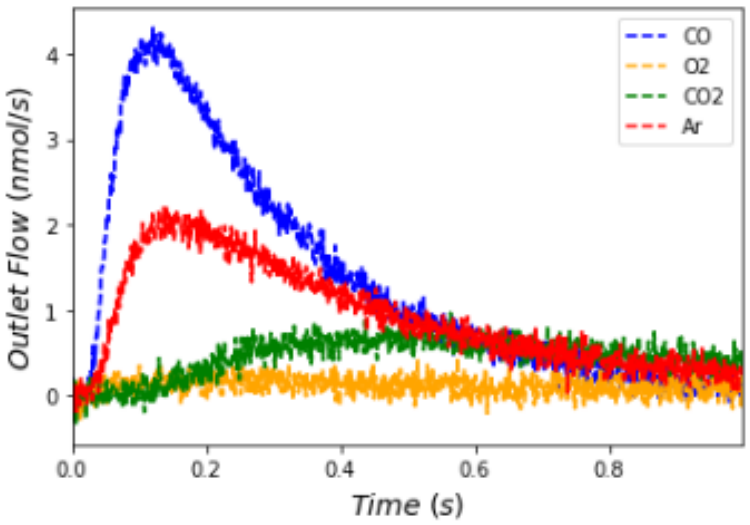

Figure 3.5

3.2 State altering experiments

State altering experiments are one of the greatest benefits of TAP, allowing you to observe the evolution of the catalyst state. These simulations can be run in a similar fashion as the state defining simulation. The only difference is in increasing the number of pulses from 1 (the second input into the $forward\_problem$ command) to however many pulses you'd like to simulate. For example, we can simulate five pulses in series by running the following.

```
forward_problem(1,5,simulation)
```

The output text follows a similar pattern as the state-defining simulation.

```
Warning: Catalyst zone will be refined and rounded to the nearest whole mesh point!

New Catalyst Fraction = 0.01
Old Catalyst Fraction = 0.009900990099009875
Change = 1.0%

Pulse #: 0
Percent: [------------------->] 99.9%Completed in: 13.324 seconds
Pulse #: 1
Percent: [------------------->] 99.9%Completed in: 16.635 seconds
Pulse #: 2
Percent: [------------------->] 99.9%Completed in: 19.39 seconds
Pulse #: 3
Percent: [------------------->] 99.9%Completed in: 20.548 seconds
Pulse #: 4
Percent: [------------------->] 99.9%Completed in: 20.675 seconds
```

Figure 3.6

3.3 Practice problem

The jupyter notebook $1a\_co\_oxidation\_problem$ offers an example problem with a solution ($1b\_CO\_oxidation\_solution$) in the github repo (linked here).

CHAPTER 4
THE INVERSE PROBLEM

To explore the optimization of kinetic parameters in TAPsolver, we are going to introduce a simple problem involving oxygen adsorption. The entire code used is provided below.

```
from tapsolver import *
new_reactor = reactor()
species = reactor_species()
O2 = define_gas()
O2.mass = 28
O2.intensity = 1
O2.noise = 0.05
O2.sigma = 0.05
species.add_gas('O2',O2)
Ar = define_gas()
Ar.mass = 40
Ar.intensity = 1
Ar.noise = 0.05
Ar.sigma = 0.05
species.add_inert_gas('Ar',Ar)
s = define_adspecies()
s.concentration = 0
species.add_adspecies('O*',s)
s = define_adspecies()
s.concentration = 100
species.add_adspecies('*',s)
mech = mechanism()
mech.elementary_processes[0] = elementary_process('O2 + 2* <-> 2O*')
mech.elementary_processes[0].forward.k = 0.1
mech.elementary_processes[0].backward.k = 2
for j in mech.elementary_processes:
  mech.elementary_processes[j].forward.use = 'k'
  try:
    mech.elementary_processes[j].backward.use = 'k'
  except:
    pass
mechanism_constructor(mech)
simulation = TAPobject()
simulation.reactor = new_reactor
simulation.reactor_species = species
simulation.mechanism = mech
simulation.output_name = 'o2_data'
forward_problem(0.5,1,simulation)
```

4.1 Fitting parameters

To perform the inverse analysis of TAP experiments, we must provide some additional details within the TAPobject. First we must specify the folder/directory we want any results to be stored. This is accomplished with

```
simulation.output_name = 'o2_fit'
```

Next, we must specify the path to the experimental data that we're fitting our mechanism with. This is done with the variable name '$data\_name$'.

```
simulation.data_name = './o2_data/TAP_experimental_data.json'
```

The next two components to include is a list of the gasses to be fitted and a list of the parameters that we want to fit. This is achieved through the adjustment of $gasses\_object$ and $parameters\_of\_interest$ variables.

```
simulation.gasses_objective = ['O2']
simulation.parameters_of_interest = ['TAPobject_data.mechanism.
                                     elementary_processes[0].forward.k','
                                     TAPobject_data.mechanism.
                                     elementary_processes[0].backward.k',
                                     'TAPobject_data.mechanism.
                                     elementary_processes[1].forward.k']
```

We only have oxygen adsorbing and desorbing and assume dissociative adsorption. For this reason, we only include oxygen in the objective function and we have two kinetic parameters in the list (adsorption and desorption rate constants). Last, we switch the 'optimize' argument to True.

```
simulation.optimize = True
```

Following all of these new specifications, we can call the $forward\_problem$ argument to run the optimization process.

```
forward_problem(0.5,1,simulation)
```

4.2 Experimental uncertainty quantification

Last, it is important to note that we can quantify the uncertainty around our fitted parameters with the inclusion of a handful of new terms. First, we want to update our parameters to the local minimum values, which is possible by calling the command $update\_parameters$. This command generates a new TAPobject with the updated parameters that can be used later. Second, like the $optimize$ parameter, must be switched to True, written below.

```
simulation_update = update_parameters(simulation)
simulation_update.uncertainty = True
```

Again, we can run the uncertainty quantification analysis by calling the $forward\_problem$.

```
forward_problem(0.5,1,simulation_update)
```

4.3 Practice problem

Example problems can be found in the TAPsolver github repo.

CHAPTER 5
CONCLUDING REMARKS

With the conclusion of this document (and the workshop), you should be able to take a kinetic model you're interested in and run a TAP simulation. You should be able to fit kinetic parameters and quantify the uncertainty of these parameters from experimental data sets. You should also be able to modify TAPobjects to match with the dimensions of the reactor or the initial conditions necessary. Although a lot of progress has been made about analyzing TAP experiments through TAPsolver, there are other tools that can be equally useful and still others that are under development. Please click the following links to learn more about two such examples:

- Click here to go to the KINNS github repo.
- Click here to go to the Tapsap github repo..

CHAPTER 6
PRACTICE PROBLEM SOLUTIONS

6.1 Chapter 2

Problem 1

```
NH3 = define_gas()
NH3.mass = 17.3
NH3.intensity = 2.5
NH3.delay = 0.1
species.add_gas('NH3',NH3)

s = define_adspecies()
s.concentration = 0
species.add_adspecies('NH3*',s)

s = define_adspecies()
s.concentration = 100
species.add_adspecies('*',s)
```

Problem 2

```
mech.elementary_processes[0] = elementary_process('NH3 + * <-> NH3*')
mech.elementary_processes[1] = elementary_process('NH3* + * <-> NH2* + H
                                    *')
```

problem 3

```
mech.elementary_processes[0].forward.k = 1
mech.elementary_processes[0].backward.k = 2

mech.elementary_processes[1].forward.k = 3
mech.elementary_processes[1].backward.k = 4
```

6.2 Chapter 3

6.3 Chapter 4